# Simultaneous Multithreading Applied to Real Time Artifact Evaluation

This document describes the experiments conducted in the paper “Simultaneous Multithreading Applied to Real Time” by Sims Osborne, Joshua Bakita, and James Anderson, to be presented at ECRTS ‘19. Table 1 (baseline task execution times), Fig. 5 (execution rates for all pairs of tasks) and all graphs in the paper can be duplicated by following these instructions.

Benchmarks were tested on an Intel Xeon Silver 4110 2.1 GHz (Skylake) CPU running Ubuntu 16.04.6. Data analysis was performed with Python 3, with some results formatted using Excel. We expect benchmarks will run on any Hyperthreading-enabled Intel platform (this includes most Xeon and Core processors) running Linux, but results will vary with a different platform. Running on a virtual machine will not produce the same results.

All files are available on the git repository https://github.com/JoshuaJB/SMART-ECRTS19.git.

## Enabling and Disabling Hyperthreading

Linux identifies processors with the following information that can be viewed by running cat /proc/cpuinfo.

Linux maintains a unique processor number for all hardware threads in the system. With Hyperthreading enabled, sibling processors can be identified as the processors which share a common physical id (i.e. socket number) and core id, but have distinct processor numbers. With hyperthreading disabled, there will be only one processor id corresponding to each unique physical id/ core id combination. Typically, sibling threads will have processor numbers that differ by the total number of physical cores (i.e. on a 16-core system, processors 0 and 16 are siblings), but this relationship should be verified prior to running benchmark experiments.

Hyperthreading can be enabled and disabled under advanced boot options. Once Hyperthreading is enabled in the boot menu, it can be turned off without rebooting by running the provided script deactivateCoresSMT.bash and enabled again by running the script activateCores.bash. Both scripts are in the folder interference-benchmark. The scripts assume a system of 16 physical cores with sibling threads having processor IDs separated by 16; if this is not the case on the target platform, the scripts will need to be edited.

## The Benchmark Programs

Table 1 and Fig. 5 in the paper report results based on our modified versions of the TACLeBench sequential benchmarks.

All files related to running benchmark programs and summarizing data are in the folder interference-benchmark.

The folder benchmarks contains the benchmarks. It consists of 23 folders plus the file extra.h, which contains macros that are added in to the benchmarks to assist with testing.

Within each benchmark folder, the main file has the name [folderName].c. We have modified the benchmarks by adding input arguments, placing the main function in a loop, and outputting timing results of each loop to a file.

Each modified benchmark accepts the following parameters.

- **maxJobs** determines how many jobs the program executes, assuming output=1 (see below).
- **thisProgram, thisCore, otherCore**, and **otherProgram** are used to make the output more readable. They are not used in any decision making; all decisions based on those variables are handled by the bash scripts which run the relevant tests.
- **runID** is used to define the name of the output file and also prints as part of the output.
- **output:** if set to 1, the program will loop maxJobs times, killing the cache at the beginning of each loop and recording the time taken for each loop after the first (in nanoseconds). The first loop is not timed, but is used to make sure all relevant data is in memory; once brought into memory all data is kept there by the mlockall() system call. At the end of each loop, outside the timer, the program saves the recorded time. At the end of maxLoops, the program will kill all programs whose PIDs were passed as input parameters and print output to the file runID.txt. If output is set to 0, the program will not time itself, will not kill the cache, and will give not output. It is expected that if output=0, maxJobs will be an arbitrarily large value and the program will be terminated by receiving a kill signal from another program.
- A string listing PIDs that should be killed upon program termination. This parameter has no effect and can safely be omitted if output=0.

## Compilation

The following bash script will compile all benchmarks and place them in the desired destination folder. It is expected that tacleNames.txt (included in interference-benchmark) is in the location from which the command is run and that extra.h is in the folder interference-benchmark. An example script is given in compileAll.sh.

```
while read t; do
        gcc /path/interference-benchmark/benchmarks/$t/*.c –o /destFolder/$t
done <tacleNames.txt
```

Note tacleNames.txt does not include all 23 benchmark programs; four of the programs--anagram, audiobeam, g723_enc, and huff_dec--were excluded from compilation and execution, since they could not be adapted to execute in a loop as we required.

## Benchmark execution

All measurements were conducted on a system that was idle apart from automatic system processes. Results may vary in the presence of other work.

To measure executing times without SMT, disable Hyperthreading and execute the command

sudo chrt –f 1 ./baselineWeighted.sh *core baseJobs runID*. (note that baselineWeighted will have to first be given execution permission by running chmod 755 baselineWeighted.sh)

in the folder to which the benchmarks were compiled (destFolder above). The destination folder needs to contain a copy of the folder tacleNames.txt. Running the script will execute every benchmark from baseJobs to 100*baseJobs times (shorter benchmarks get additional loops) on the specified core with priority fifo 97 (greater than all ordinary programs) and output the time for each run to the file r*unID*.txt. Our experiments used coreID=0 and baseJobs=1000. We recommend first performing a trial run with baseJobs=10.

To measure execution times with SMT, enable Hyperthreading and then execute the command

sudo chrt –f 1 ./allPairsWeighted.sh *firstCore secondCore baseJobs runID* where firstCore and secondCore give the coreIDs of two threads that share a physical core. We used firstCore=0, secondCore=16, and baseJobs=1000. We recommend a trial run with baseJobs=10. Execution with baseJobs=1000 may take several hours.

Doing so will create the file *runID*.txt giving for each benchmark a list of its runtimes when co-scheduled with every other job, including a second copy of itself.

### Summarizing Benchmark results

Executing the file summarize.py with input parameter *runID.txt* will output a summary of the data contained in runID.txt as a space-delimited file. Note that for the file containing baseline data, the column “second,” intended to show the interfering program, will read “none” for all tasks.

To obtain results in the same format as Fig. 5, paste the results from baselineWeighted.sh into the tab “baselineSummary” of the excel file comparison.xls and the results from allPairsWeighted.sh into the tab “threadedSummary.” In both cases, the existing data should be replaced. After calculations, coefficient of variation results appear in the tab “Co. Var.” Fig. 5 will be reproduced in tab ComparisonMax beginning in column AA.

### Analyzing Benchmark Results

To fit a statistical distribution to friendliness and strength values as we did, copy the table beginning in column AA of the ComparisonMax Excel spreadsheet. Include the top labels, but exclude the left-hand labels, the Minimum column on the right, and the Minimum row on the bottom. Paste into a new Excel sheet and save the new sheet as a file named “exp_data.csv” in the root of the repository. Run "./gen_mean_and_stdev.py" in your terminal (NumPy required). The calculated means and standard deviations will then be computed and printed to standard out.

Note: these directions apply only to our Gaussian method. The uniform method was not based on a formal statistical analysis.

## Synthetic Task Creation and Testing

To duplicate our schedulability tests and graphs, follow the following steps.

1. Make sure that Python 3, NumPy, and MatPlotLib installed and accessible from your PATH.

2. Remove existing files from the results folders, for both the gaussian-average and uniform-normal; the scripts check for existing results prior to running and will not run if results are found. To re-create our graphs using existing data, skip this step and proceed to step 5.
3. Open the gaussian-average folder in a terminal and run "./run_4-32_mixed_stdev.sh 100"
4. Open the uniform-normal folder in a terminal and run "./run_4-32.sh 100"
5. Once both scripts complete, run "./gen_graphs.py 100" from the root of our repository. This will generate and save the graphs as shown in our paper in gaussian-average/results/graphs and uniform-normal/results/graphs. Note that your results may be slightly more noisy than ours. We used 1000 samples for our plots, however, this took over a week to compute and so here we recommend running the tests with only 100 samples to keep the compute times feasible.

If you wish to conduct your own schedulability tests using different utilization ranges or different parameters for the distributions used to create execution rates, those values can be edited in the files uniform-normal/RunTests.py and gaussian-average/RunTests.py

Additional details are provided in gaussian-average/README.md and uniform-normal/README.md.